Swift/T: Scalable Data Flow Programming for Many-Task Applications

Justin M. Wozniak
Argonne National Laboratory & University of Chicago
wozniak@mcs.anl.gov

Timothy G. Armstrong
University of Chicago
tga@uchicago.edu

Michael Wilde
Argonne National Laboratory & University of Chicago
wilde@mcs.anl.gov

Daniel S. Katz
Argonne National Laboratory & University of Chicago
d.katz@ieee.org

Ewing Lusk
Argonne National Laboratory
lusk@mcs.anl.gov

Ian T. Foster
Argonne National Laboratory & University of Chicago
foster@mcs.anl.gov

1. Introduction

Many important application classes that are driving the requirements for extreme-scale systems—branch and bound, stochastic programming, materials by design, uncertainty quantification—can be productively expressed as many-task data flow programs. The data flow programming model of the Swift parallel scripting language [6] can elegantly express, through implicit parallelism, the massive concurrency demanded by these applications while retaining the productivity benefits of a high-level language.

However, the centralized single-node evaluation model of the previously developed Swift implementation limits scalability. Overcoming this important limitation is difficult, as evidenced by the absence of any massively-scalable data flow languages in current use. The primary challenge is the efficient integration of highly distributed task load balancing with global access to shared data.

We present here Swift/T, a new data flow language implementation designed for extreme scalability. Its technical innovations include a distributed data flow engine that balances program evaluation across massive numbers of nodes using data-flow-driven task execution and a distributed data store for global data access. Swift/T extends the Swift data flow programming model of external executables with file-based data passing to finer-grained applications using in-memory functions and in-memory data.

We have evaluated the performance and programmability of Swift/T for a collaboration graph analysis and optimization application, a branch-and-bound game solver, and synthetic stress tests of language constructs. Our tests show that Swift/T can already scale to 128K compute cores with 85% efficiency for 100-second tasks. Thus, Swift/T provides a scalable parallel programming model for productively expressing the outer levels of highly-parallel many-task applications. The benefits of these advances are illustrated by considering the Swift code fragment in Figure 1.


```
1  int X = 1000, Y = 1000;
2  int A[][];
3  int B[];
4  foreach x in [0:X-1] {
5  foreach y in [0:Y-1] {
6    if (check(x, y)) {
7      A[x][y] = g(f(x), f(y))
8    } else {
9      A[x][y] = 0;
10   }}
11 B[x] = sum(A[x]);
12 }
```

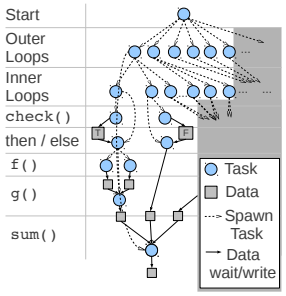

Figure 1. Simple data flow application.

The implicit parallelism of this code generates 1 million concurrent executions of the inner block of expressions, invoking as many as 4M function calls (3M within conditional logic). Previously, the single-node Swift engine would perform the work of sending these leaf function tasks to distributed CPUs at <500 tasks/sec. The new Swift/T architecture, in contrast, can distribute the evaluation of the outer loop to *many* CPUs, each of which can in turn distribute the inner loop to many additional CPUs. The diagram on the right illustrates how evaluation of the entire program — not just the external tasks at the leaves of the call graph — can utilize many nodes to rapidly generate massive numbers of leaf tasks. Tasks in this model are managed by Turbine and ADLB, described below.

2. Applications

Ensemble studies involving different methodologies such as uncertainty quantification, parameter estimation, graph pruning, and inverse modeling all require the ability to generate and dispatch tasks on the order of millions to the distributed resources. *Regional watershed analysis and hydrology* are investigated by the Soil and Water Assessment Tool (SWAT), which analyzes hundreds of thousands of data files via MATLAB scripts on hundreds of cores. This application will utilize tens of thousands of cores and more data in the future. SWAT is a motivator for our work because of the large number of data files. *Biomolecular analysis* by using ModFTDock results in a large quantity of available tasks [3], and represents a complex, multi-stage workflow.

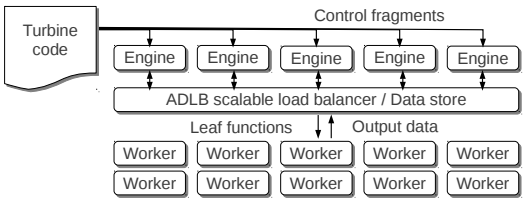

Figure 2. Architecture of Swift/T runtime: Engines evaluate Swift language semantics; workers execute leaf-task applications.

3. Programming Model

We seek to provide a system that allows code written by non-experts to run at extreme scale. This goal might be infeasible in a fully general model for parallel computation. However, we focus on many-task applications, which exhibit simpler coordination patterns but nevertheless can be challenging to scale up in commonly used message-passing programming models.

Hierarchical programming: We assume that much performance-critical code will remain in lower level languages such as C, Fortran, or even assembly, using threads or MPI for fine-grained parallelism. Data flow scripting provides a powerful mechanism for coordinating these high-performance components, as it enables fault-tolerance, dynamic load balancing and rapid composition of components to meet new application needs. In Swift, each lower-level component is viewed as a black box with well-defined inputs and outputs.

Implicit parallelism: Swift makes parallelism implicit, similarly to other data flow programming languages such as Sisal [2] and Id [5]. When control enters a code block, any Swift statement in that block can execute concurrently with other statements. This concurrent execution is feasible because of the functional nature of Swift, where we avoid mutable state and use write-once variables pervasively to schedule execution based on data dependencies. Each operation, down to basic arithmetic, can be realized as an asynchronous task, eligible to be executed anywhere in the distributed-memory computer.

For implicit and pervasive parallelism to be manageable, we need a simple model for language semantics. It has been argued [1] that parallel languages should have a deterministic sequential interpretation for most language features, with non-determinism introduced only through explicit non-deterministic constructs. All core data types in Swift, *including arrays*, are guaranteed to be deterministic and referentially transparent.

Turbine execution model: Turbine enables distributed execution of large numbers of user functions and of control logic used to compose them. Turbine requires the compiler to break user program code into many discrete ***fragments***, to enable all work to be load balanced as discrete tasks. These fragments are either user-defined ***leaf functions***, such as external compiled procedures or executables, or ***control fragments*** for data flow coordination logic. Turbine engines execute control tasks, while workers execute leaf functions, as shown in Figure 2. Execution of a Turbine control logic fragment may produce additional control fragments that are redistributed via ADLB. Turbine tracks data dependencies between tasks in order to know when each is eligible to run. Turbine provides a globally-addressable *distributed future store* [7], which drives data-dependent execution and allows typed data operations.

The Asynchronous Dynamic Load Balancer (ADLB) is an MPI library for distributing tasks (work units) among worker processes [4]. ADLB is a highly scalable system without a single bottleneck, and has been successfully used by large-scale physics applications.

Mapping Swift functions onto Turbine tasks: Computationally intensive non-Swift functions such as compiled functions or command-line applications execute as Turbine leaf functions, while control flow in the Swift language is implemented by using Turbine control tasks. If, as is often the case, control flow in a Swift function requires multiple waits for data, that Swift function must be compiled to multiple control fragments.

Limited non-determinism: Some patterns are difficult to express efficiently with write-once variables, for example, branch pruning in branch and bound algorithms and shared counters. *Updatable* variables can better support such patterns. An updatable variable is initialized to a fixed number, and can then be updated with one of several commutative update operations. The value retrieved by each read will not be deterministic, but the commutativity property makes the non-determinism more usable than a variable supporting arbitrary mutation.

Swift/T extension functions: Since Swift/T is a many-task computing language, making external code callable from Swift is crucial. Currently we support applications that call C/C++/Fortran functions from Swift scripts, by using SWIG to automatically generate wrappers.

4. Future Work

We are aware of many potential optimizations to improve Swift/T performance, such as caching, relaxing consistency, and coalescing Turbine operations at compile or run time. Garbage collection is required to support longer-running jobs that create more global data. We intend to explore other load balancing methods and data-aware scheduling and expect that advances in this area will yield many-fold improvements to Swift/T's current scalability.

Acknowledgments

This research is supported by the U.S. DOE Office of Science under contract DE-AC02-06CH11357, FWP-57810. Computing resources were provided by the Argonne Leadership Computing Facility. This material was based on work (by DSK) supported by the National Science Foundation, while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve,, M. Snir. Parallel programming must be deterministic by default. In *Workshop Hot Topics in Parallelism: HotPar'09*.

[2] J. T. Feo, D. C. Cann,, R. R. Oldehoeft. A report on the Sisal language project. *J. Parallel and Distributed Computing*, 1990.

[3] M. Hategan, J. Wozniak,, K. Maheshwari. Coasters: uniform resource provisioning and access for scientific computing on clouds and grids. In *Proc. Utility and Cloud Computing*.

[4] E. L. Lusk, S. C. Pieper,, R. M. Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, 2010.

[5] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Tech. rep., Massachusetts Institute of Technology, 1986.

[6] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz,, I. Foster. Swift: A language for distributed parallel scripting. *Par. Comp.*, 2011.

[7] J. M. Wozniak, T. G. Armstrong, E. L. Lusk, D. S. Katz, M. Wilde,, I. T. Foster. Turbine: A distributed memory data flow engine for many-task applications. In *Int'l Workshop Scalable Workflow Enactment Engines and Technologies (SWEET) 2012*.